SECTION 7.6

The Fibonacci Heap

- **push()** — $O(1)$ amortized
  Lazy, just adds singleton nodes to the rootlist

- **decreasekey()** — $O(1)$ amortized
  Does some work to keep the trees in shape
  Adds singleton nodes to the rootlist

- **popmin()** — $O(\log N)$ amortized
  Cleans up the rootlist
  (at most one tree of any given degree)
def dijkstra(g, s):
    ...
    toexplore = PriorityQueue()
    toexplore.push(s, key=0)
    while not toexplore.is_empty():
        v = toexplore.popmin()
        for (w, edgecost) in v.neighbours:
            dist_w = v.distance + edgecost
            ...
            toexplore.decreasekey(w, key=dist_w)

QUESTION. How can decreasekey be $O(\log N)$?

Doesn’t it take $O(N)$ in the first place, to find the heap node that we want to decrease?
def dijkstra(g, s):
    ...  
    toexplore = PriorityQueue()
    toexplore.push(s, key=0)
    while not toexplore.is_empty():
        v = toexplore.popmin()
        for (w, edgecost) in v.neighbours:
            dist_w = v.distance + edgecost  
            ...  
            toexplore.decreasekey(w, key=dist_w)

The first step is to implement a FibNode class to represent a node in the Fibonacci heap, and a FibHeap class to represent the entire heap. Each FibNode should store its priority key k, and the FibHeap should store a list of root nodes as well as the minroot.
SECTION 7.8
Amortized analysis of the Fibonacci Heap

decreasekey() pushes loser nodes into the rootlist

decreasekey has true cost $O(L)$ so we want $\Delta \Phi = -L$ to pay for it

Φ = num.roots + 2 × num.losers pays in advance for these “uncontrolled” iterations

popmin merges trees in its cleanup phase, true cost $O(M)$ so we want $\Delta \Phi = -M$ to pay for it

cleanup involves $M$ merges

end up with at most one tree of any given degree
SECTION 7.8

Amortized analysis of the Fibonacci Heap

SHAPE THEOREM

In a Fibonacci heap with \( N \) items, every node has degree \( \leq \log_\phi N \) where \( \phi \) is the golden ratio.

popmin also has to do \( O(d_{\text{max}}) \) work where \( d_{\text{max}} \) is the maximum possible degree in a heap with \( N \) items.
SHAPE THEOREM
In a Fibonacci heap with $N$ items, every node has degree $\leq \log_\phi N$

SHAPE LEMMA
Consider a subtree in a Fibonacci heap. If the subtree’s root has $d$ children, then the number of nodes in the subtree is $\geq F_{d+2}$ where $F_1, F_2, \ldots$ are the Fibonacci numbers
**SHAPE THEOREM**

In a Fibonacci heap with $N$ items, every node has degree $\leq \log_\phi N$

*Proof of theorem.*

Pick a node with maximum degree, call it $d$, and consider the subtree rooted at this node.

\[ N \geq \text{num. nodes in subtree} \geq F_{d+2} \geq \phi^d \]

Hence $d \leq \log_\phi N$.

**SHAPE LEMMA**

Consider a subtree in a Fibonacci heap. If the subtree’s root has $d$ children, then the number of nodes in the subtree is $\geq F_{d+2}$ where $F_1, F_2, \ldots$ are the Fibonacci numbers.
SHAPE LEMMA
Consider a subtree in a Fibonacci heap. If the subtree’s root has \(d\) children, then the number of nodes in the subtree is \(\geq F_{d+2}\) where \(F_1, F_2, \ldots\) are the Fibonacci numbers.

GRANDCHILD RULE
A node \(x\) is said to satisfy the grandchild rule if its children can be ordered, call them \(y_1, \ldots, y_d\), such that for all \(i \in \{1, \ldots, d\}\)
\[
\text{num. grandchildren of } x \text{ via } y_i \geq i - 2
\]

ALGORITHMIC CLAIM
In a Fibonacci heap, at every instant in time, every node \(x\) satisfies the grandchild rule, when we order its children \(y_1, \ldots, y_d\) by when they became children of \(x\).
SHAPE LEMMA
Consider a subtree in a Fibonacci heap. If the subtree’s root has \( d \) children, then the number of nodes in the subtree is \( \geq F_{d+2} \) where \( F_1, F_2, \ldots \) are the Fibonacci numbers.

GRANDCHILD RULE
A node \( x \) is said to satisfy the grandchild rule if its children can be ordered, call them \( y_1, \ldots, y_d \), such that for all \( i \in \{1, \ldots, d\} \)
\[
\text{num. grandchildren of } x \text{ via } y_i \geq i - 2
\]

MATHEMATICAL CLAIM
Consider a tree where all nodes satisfy the grandchild rule. Let \( N_d \) be the smallest number of nodes in a tree whose root has \( d \) children. Then \( N_d = F_{d+2} \).

\[
\text{num. nodes in tree} \geq N_{d-2} + N_{d-3} + \cdots + N_1 + N_0 + N_0 + 1
\]

\[
N_d = N_{d-2} + N_{d-3} + \cdots + N_0 + N_0 + 1
\]

\[
N_{d-1} = N_{d-3} + \cdots + N_0 + N_0 + 1
\]

\[
\Rightarrow N_d = N_{d-2} + N_{d-1}
\]

\[
\Rightarrow N_d \text{ is a Fibonacci number}
\]
SECTION 7.9
Disjoint sets
```

def kruskal(g):
    tree_edges = []
    partition = DisjointSet()
    for v in g.vertices:
        partition.add_singleton(v)
    edges = sorted(g.edges, sortkey = \lambda (u,v,weight): weight)
    for (u,v,edgeweight) in g.edges:
        p = partition.get_set_with(u)
        q = partition.get_set_with(v)
        if p != q:
            tree_edges.append((u,v))
            partition.merge(p, q)
```

AbstractDataType DisjointSet:
# Holds a dynamic collection of disjoint sets

# Add a new set consisting of a single item (assuming it's not been added already)
add_singleton(Item x)

# Return a handle to the set containing an item.
# The handle must be stable, as long as the DisjointSet is not modified.
Handle get_set_with(Item x)

# Merge two sets into one
merge(Handle x, Handle y)
Each item points to a representative item for its set
mysets = \{a:a, b:a, c:e, d:e, e:e, f:e, g:g\}
IMPLEMENTATION 1 “FLAT FOREST”

Each item points to a representative item for its set
Each set has a linked list, starting at its representative

```python
def merge(x, y):
    for every item in set y:
        update it to belong to set x

def get_set_with(x):
    return x's parent
```
IMPLEMENTATION 1 “FLAT FOREST”

Each item points to a representative item for its set
Each set has a linked list, starting at its representative

```python
def merge(x,y):
    for every item in set y:
        update it to belong to set x

def get_set_with(x):
    return x's parent
```
Sets are stored as trees
Use the root item to represent the set

```python
def merge(x, y):
    # update one of the roots to point to the other

def get_set_with(x):
    # walk up the tree from x to the root
    return this root
```

**QUESTION.** What’s a sensible heuristic for `merge`, to speed up `get_set_with`?
def merge(x, y):
    as before, using the Union by Rank heuristic

def get_set_with(x):
    walk up the tree from x to the root
    walk up again, and make all items point to root
    return this root
Can we ‘manifest’ our workings so that subsequent operations benefit?

```python
0  def selectSort(a):
1      """BEHAVIOUR: Run the selectsort algorithm on the integer
2      array a, sorting it in place.
3
4      PRECONDITION: array a contains len(a) integer values.
5
6      POSTCONDITION: array a contains the same integer values as before,
7      but now they are sorted in ascending order."""
8
9      for k from 0 included to len(a) excluded:
10         # ASSERT: the array positions before a[k] are already sorted
11
12         # Find the smallest element in a[k:END] and swap it into a[k]
13         iMin = k
14         for j from iMin + 1 included to len(a) excluded:
15             if a[j] < a[iMin]:
16                 iMin = j
17             swap(a[k], a[iMin])
```

\[
\begin{array}{cccc}
\text{A} & \text{B} & \text{C} & \text{D} \\
\text{val}=2 & \text{val}=5 & \text{val}=3 & \text{val}=1 \\
\end{array}
\]

1. Find the lowest value, and put it at the front
   - Is D.val < A.val? Yes.
   - Swap A and D

\[
\begin{array}{cccc}
\text{D} & \text{B} & \text{C} & \text{A} \\
\text{val}=1 & \text{val}=5 & \text{val}=3 & \text{val}=2 \\
\end{array}
\]

2. Find the second-lowest in [B,C,A]

we had two useful pieces of information, but we didn’t keep them i
Aggregate complexity analysis

Any $m$ operations on up to $N$ items takes $O(m + N \log N)$

[Ex. sheet 6 q. 13]

Flat Forest
(with weighted-union)

Deep Forest
(with union-by-rank)

Lazy Forest
(with union-by-rank + path compression)

$O(m \log N)$

$O(m \alpha(N))$

$\alpha(N) = 0$ for $N = 0, 1, 2$
$\quad = 1$ for $N = 3$
$\quad = 2$ for $N = 4 \ldots 7$
$\quad = 3$ for $N = 8 \ldots 2047$
$\quad = 4$ for $N = 2048 \ldots 10^{80}$
Aggregate complexity analysis

Any $m$ operations on up to $N$ items takes $O(m + N \log N)$

Flat Forest
(with weighted-union)

Deep Forest
(with union-by-rank)

Lazy Forest
(with union-by-rank + path compression)

$O(m \alpha(N))$

$\alpha(N) = 0$ for $N = 0,1,2$

$= 1$ for $N = 3$

$= 2$ for $N = 4 .. 7$

$= 3$ for $N = 8 .. 2047$

$= 4$ for $N = 2048 .. 10^{80}$
1. take a handsome stoat
2. define a graph
   *vertices on a grid, and edges between adjacent grid cells*
3. assign edgeweights
   *weight=low means vertices have similar colours*
4. run Kruskal
   and find clusters of similar colour